Structural Design

Class	**Data**	***interface* => class**
Bank	Spaces (retrieve by integer position)	*List* => ArrayList<Space>
Bank	Spaces (retrieve by group)	*Map* => TreeMap<Integer, ArrayList<Space>>
Property	Price information to buy and rent the property	*Map* => HashMap<String, Integer>
Player	Store all properties	*List* => ArrayList<Property>
Player	Store all owned properties	*List* => ArrayList<Property>
Building	Get all installments by type	*Map* => HashMap<String, ArrayList<Installment>>
ChanceCard	Get case number of card to execute	*List* => ArrayList<Integer>
CommunityCard	Get case number of card to execute	*List* => ArrayList<Integer>

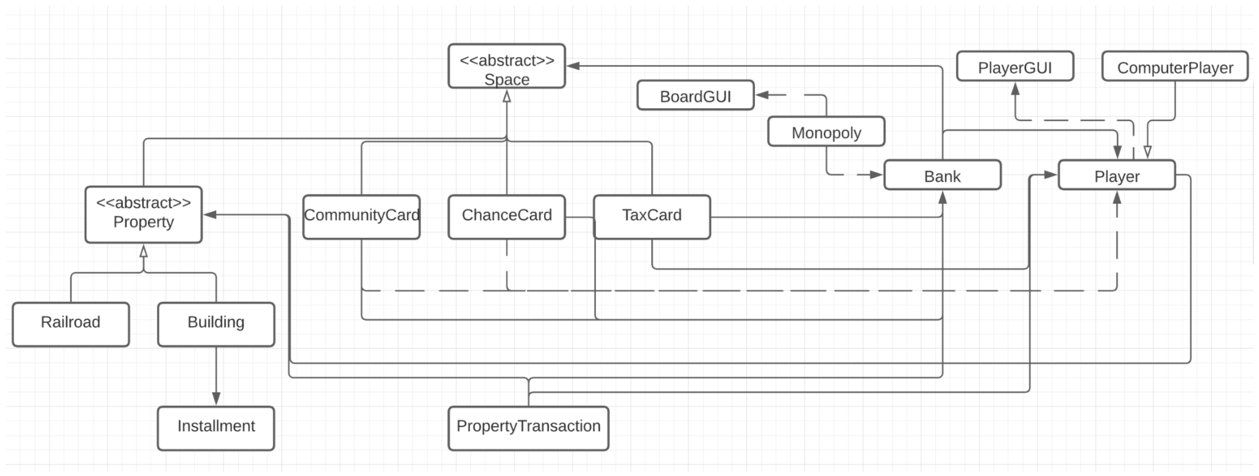
Spaces on the board must be manipulated throughout the gameplay, so multiple data structures exist to allow for the retrieval of spaces based on different properties. An `ArrayList` is best suited to retrieve spaces by their order on the board while a `TreeMap` can retrieve them by a numerical group ID. Sequential searching on the `ArrayList` representation can be used to get a property by its owner.

Meanwhile, the information for every sellable property on the board must provide easy and quick access to price information related to the cost, mortgage, and rent. This is easily accomplished through a `HashMap` with the information title as the key and the price as the value. Similarly installments, which include houses and hotels, can be accessed with a key of either “house” or “hotel” in a `HashMap`.

The `ChanceCard` and `CommunityCard` classes both utilize a random arrangement of elements, which are accessed in a set order. A `Queue` could be used, but elements would need to be added after they are used. As a result, a `LinkedList` is the more efficient implementation for this functionality.

Object Oriented Design

Classes



`Monopoly` contains the `main` method, which starts the game, and depends on `LoginGUI` and `Bank`.

`BoardGUI` provides a GUI interface for the user to add players to the game, start the game, and provide in-game information to all of the players.

`Player` represents the internal information of a player and depends on `PlayerGUI`. Specifically, this class contains the functionality to add and remove properties, manage balance, and move it's position.

An instance of `ComputerPlayer` is similar to `Player`, but it contains a risk-benefit decision making analysis instead of asking a human for input.

`PlayerGUI` creates a window for the human or computer player to interact with the program. It contains a control panel with a dropdown menu containing information relevant to the action selected. There are two buttons for the player to accept or deny an action, and a number field to enter numerical input.

The `Bank` class controls the majority of actions performed in the game. This includes adding players, creating the board, maintaining a directory of the spaces on the board, and communicating with all the players. In other words, it is a central hub for the majority of actions that involve more than one class.

The `Space` class is abstract and represents any space on the board. The class has abstract methods to retrieve the group of the property, name, and position on the board. It also implements `Comparable<Space>` to compare two spaces by position.

`CommunityCard`, `ChanceCard`, and `TaxCard` all represent actions that directly affect the player, so it is associated or dependent on the player. `CommunityCard` and `ChanceCard` randomize the order of cards at the start of the game, and implement a circularly linked list to retrieve cards. `TaxCard` can be of one of two slightly differing types, which is controlled by a field.

`Property` is an abstract class that represents a space that can be bought, sold, and create rent revenue. It has a HashTable with information about the rent, sell price, etc . . ., which provides easy access to that information when the program needs it.

`Railroad` is similar to `Building`, but the `Building` class provides the ability to add houses and hotels to the property.

A `PropertyTransaction` is created to sell or buy `Property` objects, and all instances are passed through a `Bank` instance to be processed.